A Multi-Layered Image Cache for Scientific Visualization

E. LaMar, V. Pascucci

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

April 11, 2003

A Multi-Layered Image Cache for Scientific Visualization

Eric LaMar and Valerio Pascucci

April 11, 2003

Abstract

We introduce a multi-layered image cache system
that is designed to work with a pool of rendering
engines to facilitate a frame-less, asynchronous ren-
dering environment for scientific visualization. Our
system decouples the rendering from the display
of imagery at many levels; it decouples render fre-
quency and resolution from display frequency and
resolution; allows asynchronous transmission of im-
agery instead of the compute-send cycle of standard
parallel systems; and allows local, incremental re-
finement of imagery without requiring all imagery
to be re-rendered.

Interactivity is accomplished by maintaining a
set of image tiles for display while the production
of imagery is performed by a pool of processors.
The image tiles are placed in fixed places in camera
(vs. world) space to eliminate occlusion artifacts.
Display quality is improved by increasing the num-
ber of image tiles and imagery is refreshed more
frequently by decreasing the number of image tiles.

keywords: image cache, impostors, scientific vi-
sualization, multiresolution techniques, hierar-
chical techniques, parallel techniques.

1 Introduction

Scientists are faced with a problem that as their
simulations grow to sizes where interesting features
and structures can be resolved, the features them-
selves become to small (relative to the size of the
data) to find, and the structures too large to vi-
sualize. Interactive navigation and exploration of
these datasets is essential, and small features can
only be properly understood is shown in the con-
text of larger structures, showing both large scale
structures and small scale features in the same vi-
sualization is essential. However, the data size pre-
vents efficient rendering of even single frames, let
alone multiple frame per a second that is required
for interactive exploration.

Caching and reusing imagery over several frames
to amortize the cost of rendering that imagery is
proving to be a very useful technique. However,
prior techniques are unable to meet he demands
of scientific visualization in that they expect static
scenes, require significant preprocessing time and
user assistance to place impostors, and rely on pure
random chance to catch moving structures.

A significant number of scientific datasets are
uniform, cartesian grids, where the information
density is very high and uniform. Thus, tile den-
sity must be corresponding very high and uni-
form. Scientific visualization methods require user-
controlled, run-time parameters that can signif-
icantly affect visualization results; *i.e.*, transfer-
functions for volume visualization, iso-value for iso-
contouring, or temporal movement for time-varying
datasets. All of these require adaptive refinement
of tiles and dynamic updating of these tiles.

We have developed our Multi-Level Image
Caching (MLIC) system to cache rendered imagery
and to address issues specific to the use of impos-
tors for scientific visualization. Impostors in our
system are called tiles, and associate a square im-
age with spatial position and extent (and temporal
position and extent for time varying datasets). A
set of image tiles are maintained with a single fore-
ground process displaying the imagery and a set
of parallel processes rendering the imagery. Dis-
play quality is improved by increasing the number
of tiles and update frequency is improved by de-
creasing the number of tiles. Local refinement of
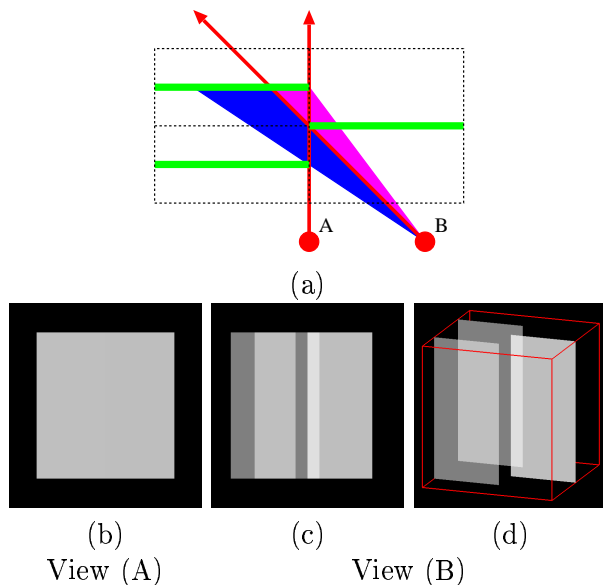data does not require re-rendering all image tiles,

(a)

(b)　　　(c)　　　(d)

View (A)　　　View (B)

Figure 1: Occlusion artifacts occur when tiles generated to be correct from viewpoint *A* are viewed from viewpoint *B*. In figure (a), viewpoints are red, tiles are green, the hole is blue, and the occlusion is magenta. Figures (b)-(d) shows figure (a) from the point-of-view; the two left tiles have opacities of 50% (which composite to opacity of 75%), and the right tile has an opacity of 75%

only those tiles that intersect the refinement region need to be re-rendered.

Our system decouples generation of imagery from the display of the imagery, and decouples the resolution of rendered imagery from the resolution of displayed imagery. Secondly, our system also decouples the placement of tiles from the specifics of the data layout; that is, tiles are placed to reflect user interest and rendering concerns.

Densely placed tiles that move with respect to the camera viewpoint commonly experience occlusion artifacts; e.g., tiles that are supposed to be adjacent to each other can appear to move apart, allowing a hole to form where structures that are supposed to be hidden are exposed, or adjacent tiles move over and occlude each other, as shown in figure 1. While these problems have been addressed in a very basic way for opaque structures, there are not prior techniques to handle transparent structures.

Our solution is to decompose the space around the point-of-view into a set of convex, non-occluding (with respect to the point-of-view) polyhedron, which are then subdivided by a k-D tree to allow for adaptivity. The k-D tree based subdivision of the polyhedron is constrained to eliminate occlusion artifacts. Tiles are placed at the nodes of the k-D tree and move with the point-of-view.

Our system runs on a two-processor linux workstation and a large symmetric multiprocessor (SMP) machine. The implementation discussed in this paper uses a cube, centered about the point-of-view, decomposed with six pyramids, where the tops of meet at the point-of-view and the bases form the six faces of the cube. Other configurations are possible.

2   Related Work

Bethel *et.al.* [BSL+00] discuss the Visipult system - a distributed, multiresolution visualization system for time-varying uniform, rectilinear datasets. Imagery is produced at the brick-level and is, therefore, not independent of the data decomposition.

There is significant amount of work [CTFB01, CSKK99, SS96, SLS+96, DSSD99] on use and preprocessing of impostors for viewing extremely large CAD datasets. Most typical applications are architecture walk-throughs, either of individual buildings, of whole cities or city districts. Significant preprocessing (with user assistance) effort is required to find good locations to place impostors and to segment the model (with respect to the impostor's location) to near and far sets. Far geometry is rendered and cached with that impostor. At runtime, near geometry is rendered directly and the far geometry is approximated with the caching imagery. These datasets static and are intended to be visualized many times, so it is reasonable to spend a large amount of preprocessing time to accelerate the rendering of them.

The have been numerous of specialized hardware solutions to parallelize rendering of datasets. Blanke *et.al.* [BBFZ00] introduces the Metabuffer, a cross-bar compositing network of *N* COTS (Common Off The Shelf) rendering machines to *M* display devices, where the transmitted imagery includes color, alpha, and depth values, so it is possible to composite mutually occluding image tiles.

2

The network can be configured to place imagery at any arbitrary location and extent on the display devices. The hardware has not been built; their results are derived from simulations.

Stoll *et.al.* [SEP$^+$01] introduce the Lightning-2 digital video-based compositing network, also connecting N COTS rendering machines to M display devices. Adaptive placement of imagery is accomplished by encoding image location in the video signal. Lightning-2 uses digital video, so communicating auxiliary (*i.e.*, depth) information in the video signal is difficult. Lombeyda *et.al.* [LMS$^+$01] introduces the Sepia-2 compositing network. This system connects N COTS rendering devices in a daisy-chain, with the output of one device is fed into the input of the next in chain, and the final device is the display. There is no adaptivity, and entire frames are composited pixel-to-pixel. The network uses a central crossbar, and can reconfigure the network with each frame. Blanke's and Stoll's systems do not scale linearly with increasing input and output devices as their network complexity is $N \times M$. While Lombeyda's system scales linearly with respect to rendering devices, but there is a small latency at each node, which restricts the total frame rate. The SGI Onyx and Origin SMPs (symmetric multi-processor) systems, equipped InfiniteReality [MBDM97] rendering engines, are a more general solution to parallel rendering. While the Onyx and Origin have a general purpose interconnect fabric that extremely fast, the InfiniteReality engine is significantly out-performed by newer cards; the Onyx and Origin systems are also limited in the total number of InfinteReality engines they can hold. The PixelFlow [EMP$^+$97] is a very specialized (very non-COTS) machine that uses a deeply pipelined compositing network connected to a set of rendering engines. Data is distributed over the rendering engines, which compute a full frame, and ship the frame to the compositing network. These hardware techniques show reasonable speed-ups for small numbers of rendering engines, displays, and datasets. However, all share the basic restrictions of being extremely expensive, very specialized, have limited adaptivity with respect to the rendered image size, and scaling is only good if one of the dimensions (rendering engines, display

size, and dataset size) is increase, but not if all increase. These systems have an explicit notion of frames and are not tolerance of delays or stalls in rendering engines. The faster systems composite digital video signals, which makes it difficult to include depth information. There are two recent commercial products, SGI's InfinitePerformance [SGI] and HP's SV6 [HP], which use a larger number of rendering nodes, connected by a compositing network. While neither are COTS, both seem to be more commodity- and component-oriented than prior machines offered SGI and HP.

There are several software techniques to decompose a scene into layers, then display the layers from a different, but limited set of alternate viewpoints. Mueller *et.al.* [MSHC99] allows transparent volume visualization, but must keep track of all composited depth values, so it can place the layers such that no gaps appear when viewing the layers from new viewpoints. Schaufler [Sch98] uses multiple layers to render fully opaque models. Gaps are avoided by overlapping the spatial extents of the layers such the the images overlap by several pixels. Layers are re-rendered just before a gap is predicted to appear. His prediction mechanism only characterizes error with respect to camera translations and not with camera rotations about the model. Shade *et.al.* [SGHS98] discuss techniques to use layers of images (with color and depth) to approximation complex objects. Images are reprojected on a pixel-by-pixel basis and require a well define depth value; this this technique is only useful for fully opaque objects.

The Tapestry project by Simmons *et.al.* [SS00] renders a scene by drawing a set gouraud-shaded triangles where the color and depth of the vertices are calculated by a ray-tracer. Triangles are refined if they are physically large or have large changes in color or depth. As the user moves, sample points (vertices) that become occluded are removed from the mesh. Their technique does not handle transparent volumes as it assumes opaque surfaces and maintains only one layer of sample points.
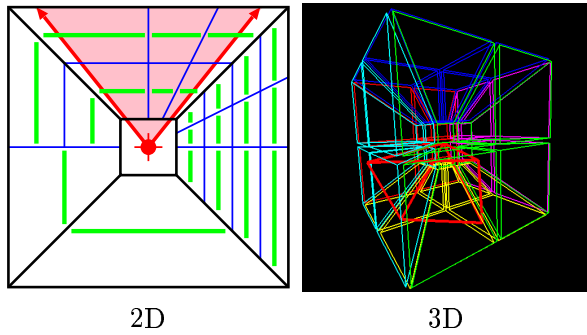
2D   3D

Figure 2: Example of a MLIC implemented on a square (2D) and a cube (3D).

(A)   (B)   (C)

Figure 3: A k-D tree tile can be subdivided in two orientations in 2D (3 in 3D).

3 MLIC Spatial Decomposition and Caching Basis

The MLIC system decomposes the region about the camera's point-of-view into a covering set of convex, non-occluding (with respect to the point-of-view) polyhedra. Many decompositions of the space around the camera are possible; in this paper, we use a cube basis for the sake of simplicity. The cube is then decomposed into six pyramids, with the "top" point at the center of the cube, and the pyramid bases forming the six faces. The six pyramid are, in turn, subdivided by constrained k-D trees.

The image tiles are placed at fixed positions with respect to the camera point-of-view, so must be re-rendered when the point-of-view translates; they don't need to be re-render if the point-of-view's orientation/direction or field-of-view change.

Figure 2 shows a 2D and 3D example of a Multi-Level Image Cache implemented using a square/cube as a basis. For figure 2(2D), the red circle at the center is the camera point-of-view, with the red lines showing the camera orientation and field-of-view; black lines show the base decomposition of the space; blue lines show the k-D tree decomposition of each quadrant; and green lines show individual tiles. The tiles are shown slightly smaller that their physical extent (delimited by blue & black lines), as to emphasize that they are independent entities. Note that the different quadrants have different degrees of decomposition. Figure 2(3D) shows each of the six faces of the cube are shown in different colors. The pyramids are scaled
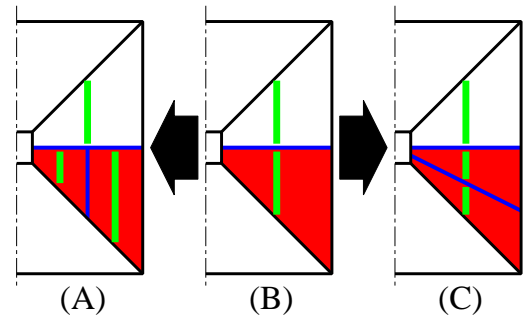
down by 10% to show the individual regions. All faces have been subdivided twice.

The distance to the outer face of a pyramid corresponds to the far-clipping plane of a viewing-frustum. Increasing and decreasing the distance to the far-clipping plane increases and decreases the spatial extent covered by the cache.

The k-D tree decomposition planes are either parallel to the cube face or pass though the origin of the cube-cache, so there are no occlusion artifacts (compared to figure 1); *i.e.*, tiles run to the boundaries of a k-D node and their end-points do not move with respect to the camera. This is shown in figure 3. Figure 3(B) shows the parent tile to be subdivided in red; figure 3(A) shows the subdivision of the parent tile into front and back tiles (with respect to the point-of-view) and figure 3(C) shows the subdivision parent tile in to left and right tiles.

Figure 4 shows where tiles would be reused between frames. In the top row, the first (left) frame, a set of tiles are rendered. The blue and green tiles are rendered for (or before) the first frame. In the second frame (top row, right), as the camera (shown in red) turns clockwise, an additional set of tiles are now visible. Those shown in green are rendered in both frames (i.e., rendered in the first frame and just reused in the second). Tiles shown in blue are not visible in the second frame, and may be deleted (if running out of cache space). The purple tiles in the second frame are now visible; they will be placed in a *work* queue to be rendered if their imagery is invalid. The middle row shows a rendering of a iso-surface rendering of a Trebecular bone dataset; the initial camera position is show in
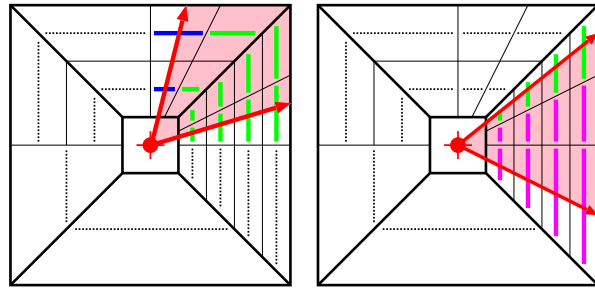
2D Example
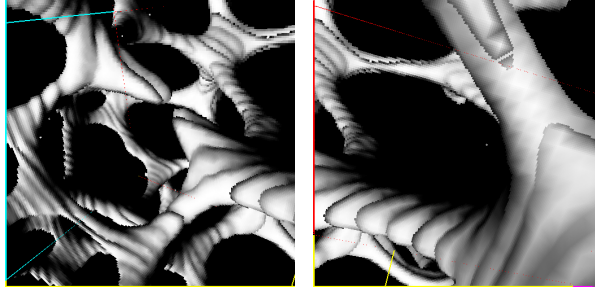
From the point-of-view
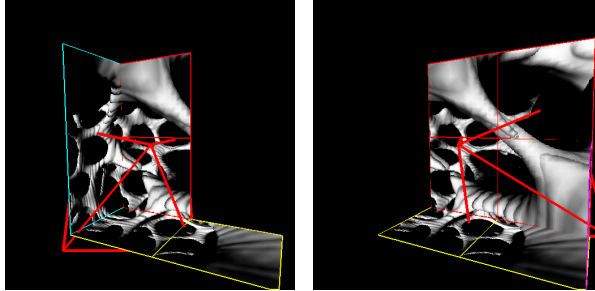
Outside, looking at point-of-view (red pyramid)

Figure 4: Reusing tiles as the camera turns to the right.

the left image and the camera has turned to the right in the right image. The bottom row shows an outside view of the MLIC, with the point-of-view and Field-Of-View pyramid shown in red. Tile boundary color corresponds to the faces of the cube for defines the physical cache configuration. Notice that the tiles outlined in cyan no longer appear in the right image, and new tiles appear on the right side of the image.

All tiles have the same fixed resolution. A tile is refined by replacing it with a left/right, top/bottom, or front/back tile children, where each child contains a copy of the corresponding region of the parent tile. The new tiles are marked for future re-rendered. Tiles are coarsened by removing the children and replacing the parent's imagery by a
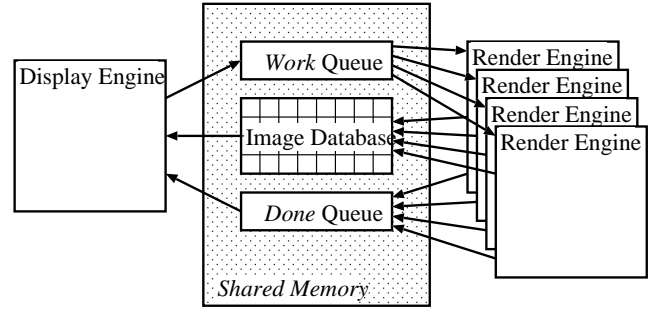

Figure 5: The MLIC system architecture.

filtered version of the children's imagery.

Rotating the camera direction does not invalidate any of the tiles, but newly exposed tiles may require rendering, and possibly refinement. Tiles now partially exposed by to be coarsened. Our current system does not use any error metric for prioritizing tiles for re-rendering and visible tiles are re-rendered in a round-robin scheme.

4 The MLIC System Architecture

The MLIC system architecture consists of a single display engine, multiple rendering engines, a *work* queue, a *done* queue, and an image database. The *work* and *done* queue and the image database are placed in a shared memory segment and the display engine and all rendering engines can directly access the queue and image database.

The display engine writes requests to to *work* and receives acknowledgment on the *done* queue. Render engines read requests from the *work* queue, renders and writes imagery to the image database, and writes completed tasks to the *done* queue.

We currently use VTK's off-screen rendering capability and its ray-cast engines (both volume rendering and iso-surface rendering) to render imagery.

The display engine just renders tiles that are both within budget and are available. If an tile has no children, but does not meet the rendering requirements, it is added to the *work* queue to be subdivided. The display engine also reads the *done* queue to see what regions have been completed by the render engines. When a region is completed, the display engine notes this in the image database. This operation is extremely small and fast, and if
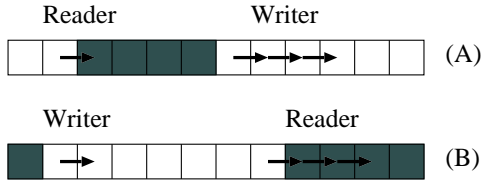
5

Figure 6: Readers and writers can compare *head* and *tail* positions to get a conservative estimate if a queue is *empty* or *full*.

the generator engines performed it, would require another semaphore control access.

4.1 Shared Queue Issues

We started with a very basic circular queue implementation for the *work* and *done* queues, using two fields to record the *head* and *tail* positions, a *size* field to record the available space, and one semaphore for each queue. Both display engines and render engines must block to add or remove elements from the queues. We found that this basic implementation scaled poorly; when using eight render engines and one display engine, processes would occasionally (~1% of the time) block for as long as 0.5 seconds on the semaphore to access the queue. This caused unacceptable stalls in the display engine.

Our solution is to observe that when using queues, where one process only writes to the queue and the other process only reads from the queue, a reader never modifies the *head* of the queue, and a writer modifies the *tail* of the queue, and that a reader can only read from a non-empty queue and a writer can only write to a non-full queue. We can conservatively estimate the empty and full status of a queue by calculating the distance between the *head* and *tail* of the queue. That is, a reader may find the queue empty, even though a writer may just added an element; similarly, a writer may find the queue full, even though a reader has just removed an element. When the reader access the queue's tail pointer and compares the distance to the head pointer, the *head* may move, increasing (never decreasing) the number of elements in the queue, thus will get a conservative estimate of available elements in the queue. A similar argument work for a writer: reader processes will only in-

crease available space in a queue. This is shown in figure 6. However, parallel readers must share and block on access the tail pointer, and parallel writers do the same for the head pointer. Hence, the display engine never blocks on either the *work* or *done* queues as there is only one; however, render engines must block on both queues. As the engines take considerable longer to complete their tasks than the display engine, they check the queues much less often. This does imply, however, that there is a scaling limitation on the render engines, and that a multi-stage or distributed queuing system will be required to scale beyond a certain point.

The only caveat is that accessing integer values on shared memory system must be atomic, and no partial values are returned. This model of a shared queue works correctly on a two processor Linux boxes and a 48 processor SGI Origin3000.

4.2 Tasks

Three kinds of tasks can be written to the *work* queue: subdivide, render, and merge. In addition to the kind of task, the task structure also contains a pointer to the associated k-D node, which includes the spatial position and extend of the region to be modified. When a render engine starts to service a task, it removes the task from the *work* queue. When a task is completed, it is simply written to the *done* queue.

4.2.1 Subdivide

The *subdivide* task is to take a tile and replace it with two children tiles that cover the same extent. An tile can be split (with respect to the point-of-view) into left/right, top/below, or front/back tile pairs. Since all tiles have the same resolution; when a left/right or top/bottom pair is rendered, the effect is to double to number of pixels in the direction of the split.

Tiles are displayed in back to front order with blending to affect an OVER operation (see [PD84]). So to break a tile into a front/back tile pair, we need to set the values in the front/back pairs such that when they are composited together, they produce the same result as displaying the parent tile. Hence, to subdivide a tile, we must compute a inverse to

the OVER operator. The OVER operator is defined as follows:

$$K \;=\; C_0 \, over \, C_1 \;=\; C_0 + C_1(1 - \alpha_0)$$
$$\gamma \;=\; \alpha \, over \, \alpha \;=\; \alpha_0 + \alpha_1(1 - \alpha_0)$$

Where $C_{1,2}$ and $\alpha_{1,2}$ are the input opacity-weighted color and opacity values, respectively, and K and γ are the output opacity-weighted color and opacity values, respectively. Note that the OVER operator performs the same calculation on each of the red, green, and blue channels. K and γ correspond to the color and opacity of the parent tile, and $C_{1,2}$ and $\alpha_{1,2}$ correspond to the color and opacity of the child tiles. We also simplify by assuming the front/back imagery is the same, e.g., $C = C_1 = C_2$ and $\alpha = \alpha_0 = \alpha_1$.

Solving for C, given K:

$$K = C + (1 - \alpha)C = C(2 - a)$$

thus

$$C = \frac{K}{(2 - \alpha)}$$

And solving for α, given γ:

$$\gamma = \alpha + \alpha(1 - \alpha) = 1 - (1 - \alpha)^2$$

thus

$$\alpha = 1 - \sqrt{1 - \gamma}$$

However, we have observed two issues with these formulas. First, these operations are performed on 8-bit integer values, and can be very error prone. Secondly, the structures in the data are not placed uniformly in the associated region. If, for example, the front half of the region associated with the parent tile is empty, the child tile associated with that front half will contain imagery belonging to the back half. This is not a problem until one of the children is re-rendered. If the front child is re-rendered, the overall contribution of the front/back regions will decrease as they effectively become more transparent. Similarly, if the back tile is re-rendered first, the overall contribution will increase, and they effectively become more opaque. Due to proximity and viewing parameters, the front and back pairs are re-rendered at nearly the same time, and therefore do not create a problem.
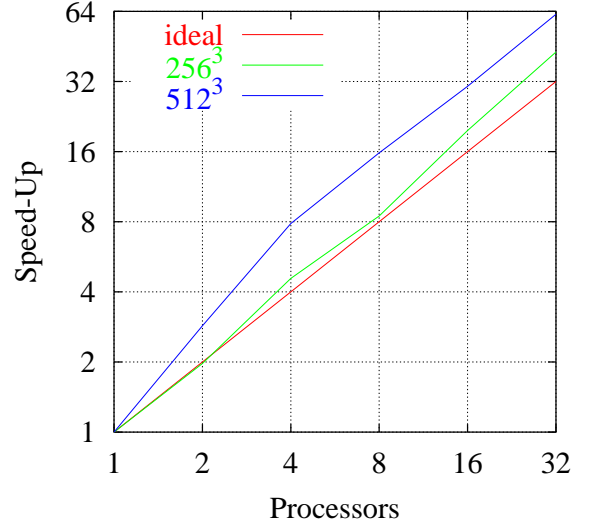

Figure 7: A log-log plot of the scalability study for two trebecular bone datasets using the Multi-Level Image Cache system.

4.2.2 Merge

Merge a left/right, top/bottom, or front/back pair for form a lower resolution image. This is used when the camera position has not changed, but the region is less important (*e.g.*, the user has turned the view frustum away). The left/right and top/bottom pairs are produced by low-pass filtering or sub-sampling, depending on which filtering mode used in the original rendering. Merging a front/back pair is simply compositing the front and back pairs together.

4.2.3 Render

Simply render a region to an image. VTK is used as the rendering engine, so anything that VTK can render can be rendered and cached in our system.

5 Results

The results were obtained using a SGI Origin3000 computer with 48 250MHz MIPS R10K processors. We use a trebecular bone dataset, which an extremely high-resolution scan of the spongy material inside of bones; the original 540^3 dataset is $1 \, cm^3$. We measured the sustained rate at which render

	256^3		512^3	
Processors	Rate	Speed-Up	Rate	Speed-Up
1	1.48	1.00	0.73	1.00
2	2.91	1.97	2.09	2.86
4	6.76	4.57	5.74	7.86
8	12.5	8.47	11.5	15.8
16	29.2	19.7	22.3	30.5
32	63.3	42.8	45.4	62.2

Table 1: Scalability study for two trebecular bone datasets using the Multi-Level Image Cache system. The rates are tiles per second.
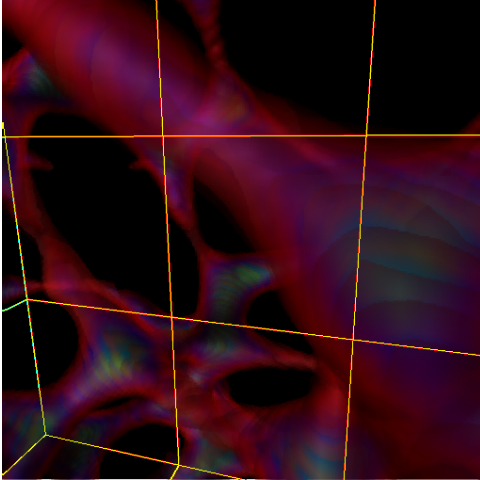

Figure 8: Volume rendering a 256^3 trebecular bone dataset using the Multi-Level Image Cache system. Individual tiles are outlined in yellow.

tasks are performed by counting the number completed during a fixed interval of 10 seconds. We did not measure subdivide or merge tasks, as they happen much less often, do not occur in a sustained fashion, and take much less computational effort than rendering.

Our scalability study was performed using our MLIC system, VTK's off-screen rendering capability (via Mesa), and VTK's software-based ray-cast iso-surface engine to render imagery. The image tiles all have a resolution of 128^2 pixels.

The MLIC volume renderer experiences roughly linear speed-up for a 256^3 and 512^3 versions of the trebecular dataset, as shown in table 1 and figure 7. The apparent super-linear speed-up for MLIC is due to the range of rates for different number of processors: runs with one processor show much larger range in the number of tiles rendered per a second than runs with 32 processors. Some tiles are more expensive to render than others; tiles that are more square touch less memory than tiles that are long trapezoids. The rates for runs with smaller numbers of processors should probably be higher. As each processor has its own copy of the data, there should be no super-linear scaling due to data fitting into the processor caches.

6   Conclusion and Future Work

We have introduced our Multi-Level Image Caching system. It maintains a set of image tiles for interactive display while a set of parallel processes render new imagery. Our system has successfully decoupled rendering rates from display rates, allows for fairly slow rendering, while providing interactive display of imagery. The system experiences linear scaling on an Origin3000 SMP up to 32 processors.

Currently, our system uses a round-robin approach to updating tiles. This is not sufficient when there are a large number of tiles being displayed. If there are a large number of tiles, the rate at which tiles are update is slow, and it can become difficult to navigate an environment. We plan to implement an error- and view-driven scheduler. We plan to replace the rendering engine. There is considerable overhead for rendering a frame in VTK, and it is not optimized for texture-based volume visualization. Third, we plan to augment the system to handle multiresolution, time-vary datasets. Last, we plan to move our system to cluster of Linux workstations, as contemporary graphics cards are much faster than the Origin3000's InfiniteReality3 graphics subsystem.

Acknowledgments

References

[BBFZ00]   W. Blanke, C. Bajaj, D. Fussell, and X. Zhang. The Metabuffer: A Scalable Multiresolution Multidisplay 3D Graphics System Using Commodity Rendering Engines. Technical Report TR2000-16, University of Texas at Austin, 2000.

[BSL⁺00]   W. Bethel, J. Shalf, S. Lau, D. Gunter, J. Lee, B. Tierney, V. Beckner, J. Brandt, D. Evensky, H. Chen, G. Pavel, J. Olsen, and B.H. Bodtker. Visapult - Using High-speed WANs and Network Data Caches to Enable Remote and Distributed Visualization. In *Super Computing 2000*, pages 118–119, November 4-10 2000.

[CSKK99]   B. Chen, J.E. Swan, II., E. Kuo, and A.E. Kaufman. LOD-Sprite Technique for Accelerated Terrain Rendering. In *IEEE Visualization 1999*, pages 291–298, October 1999.

[CTFB01]   M. Carrozzino, F. Tecchia, C. Falcioni, and M. Bergamasco. Image caching algorithms and strategies for real time rendering of complex virtual environ-

ments. In *Afrigraph 2001*, pages 65–74, November 5–7 2001.

[DSSD99]   X. Decoret, F. Sillion, G. Schaufler, and J. Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3):61–73, September 1999.

[EMP⁺97]   J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. PixelFlow: The Realization. In *Graphics Hardware Symposium*, pages 57–68, 1997.

[HP]   HP. Visualization Center SV6, http://www.hp.com.

[LMS⁺01]   S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich. Scalable Interactive Volume Rendering Using Off-the-Shelf Components. In *IEEE PVG 2001*, pages 115–121, 158, October 2001.

[MBDM97]   J.S. Montrym, D.R. Baum, D.L. Dignam, and C.J. Migdal. InfiniteReality: A Real-Time Graphics System. In *Siggraph 1997*, pages 293–302, August 3–8 1997.

[MSHC99]   K. Mueller, N. Shareef, J. Huang, and R. Crawfis. IBR-Assisted Volume Rendering. In *Hot Topics, Vis 1999*, pages 1–4, October 1999.

[PD84]   T. Porter and T. Duff. Compositing Digital Images. In *Siggraph 1984*, pages 253–259, July 1984.

[Sch98]   G. Schaufler. Per-Object Image Warping with Layered Impostors. In *Rendering Techniques 1998*, pages 145–156, 1998.

[SEP⁺01]   G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan. Lightning-2: A High-Performance Display Subsystem for PC Clusters. In *Siggraph 2001*, pages 141–148, August 12–17 2001.

[SGHS98]   J.W. Shade, S.J. Gortler, L. He, and R. Szeliski. Layered depth images. In *Siggraph 1998*, pages 231–242, July 1998.

[SGI]   Inc. SGI. InfinitePerfomance, http://www.sgi.com.

[SLS⁺96]   J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical Image Caching for Accelerated Walk-throughs of Complex Environments. In *Siggraph 1996*, pages 75–82, August 4-9 1996.

[SS96]   G. Schaufler and W. Stürzlinger. A three-dimensional image cache for virtual reality. In *Eurographics 1996*, 1996.

[SS00]   M. Simmons and C.H. Séquin. Tapestry: A Dynamic Mesh-based Display Representation for Interactive Rendering. In *Rendering Techniques 2000*, pages 329–340, June 2000.